

Sorting

Pablo Castro
Algoritmos I-UNRC

Sorting

Sorting es la tarea de organizar una colección de datos según un orden dado

- Podemos realizar sorting sobre cualquier tipo con un orden: **int**, **char**, **Integer**, **Strings**, etc
- En general, el sorting acomoda los elementos de forma ascendente o descendentes.
- Es importante que los algoritmos de sorting sean **eficientes** ya que en la práctica se quiere ordenar una cantidad grande de elementos

Sorting en JAVA

En Java se utiliza la clase COMPARABLE:

- Toda clase con un orden hereda de comparable.
- Permite implementar algoritmos de sorting polimorficos.
- La clase comparable provee un método CompareTo():

CompareTo(T o): Compara el objeto actual con **o**, retorna -1,0,1 dependiendo si **o** mas grande, igual o más chico que **this**, respectivamente.

Importante...

Para analizar un algoritmo de sorting podemos tener en cuenta:

- **Eficiencia:** el tiempo de ejecución del algoritmo, en el peor caso, y también en el caso promedio.
- **Cantidad de comparaciones:** Cuanta veces comparamos para ordenar los elementos.
- **Cantidad de Intercambios:** Cuanta intercambios se realizan para ordenar

Las comparaciones pueden ser costosas

Los intercambios son constantes en JAVA

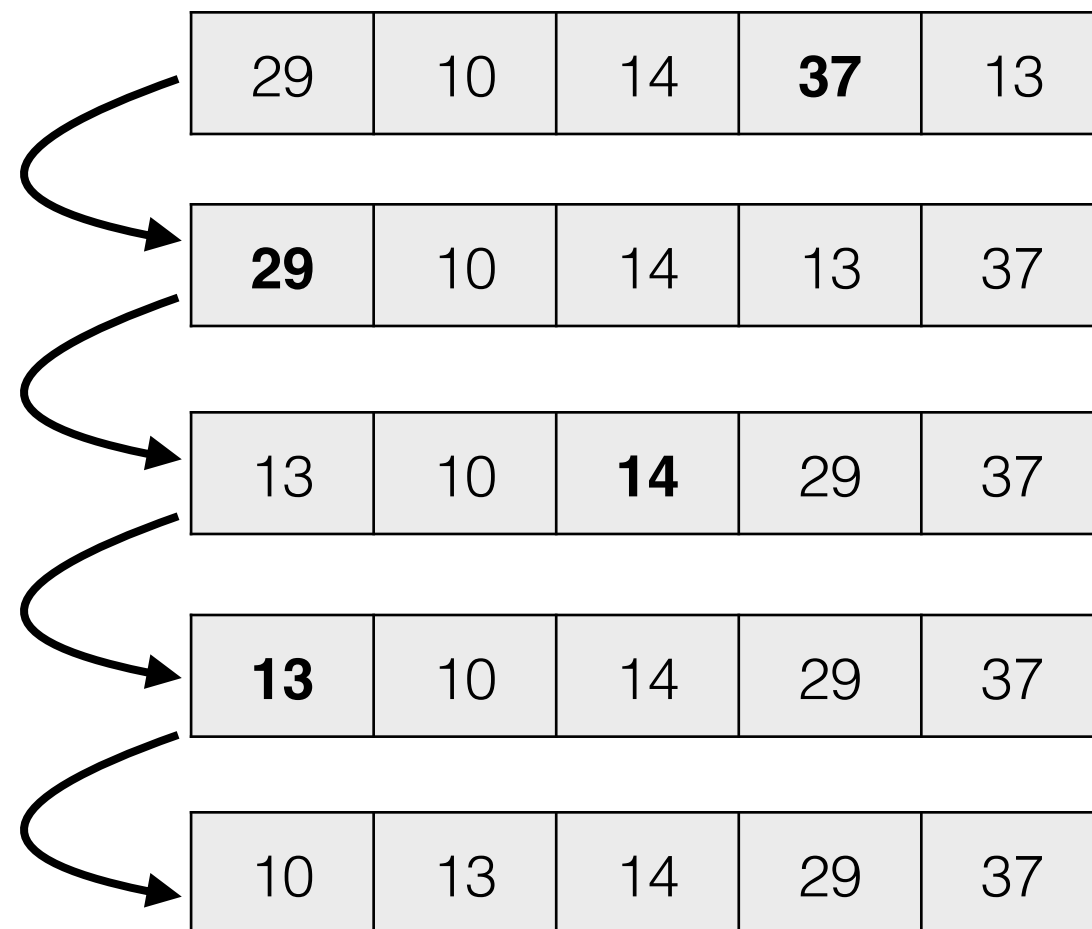
Estabilidad

Un algoritmo de sorting se dice estable si preserva el orden de los elementos con las mismas claves

- La clave es el campo o el atributo sobre el cual ordenamos.
- Hay algoritmos que son estables y otros no.
- En general cualquier algoritmo se puede hacer estable con algún costo extra: agregar más claves, etc.

Selection Sort

Idea: seleccionar el item más grande, ponerlo último; agarrar el segundo más grande, ponerlo penúltimo, etc.



Algoritmo en JAVA

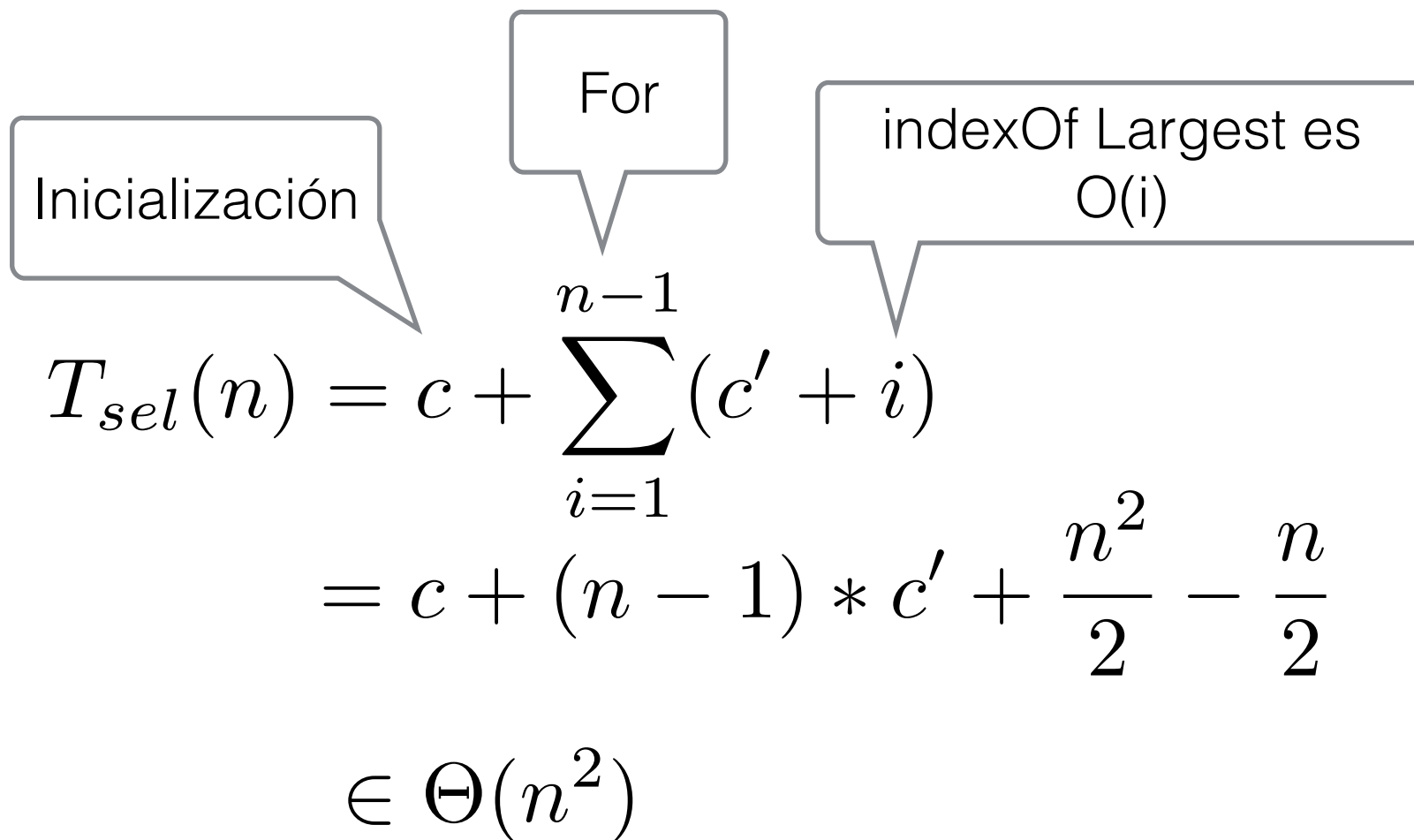
```
public static void selectionSort(Comparable[] array, int n){  
    // last: index del ultimo elemento de la parte no ordenada  
    // largest: posicion del elemento mas grande  
    for (int last = n-1; last >= 1; last--){  
        //inv: array[last..n-1] está ordenado  
        int largest = indexOfLargest(array, last+1);  
        swap(array, last, largest);  
    } // end for  
} // end selectionSort
```

En donde:

```
private static int indexOfLargest(Comparable[] array, int n){  
    int largest = 0;  
    for (int i = 1; i < n; i++){  
        if (array[i].compareTo(array[largest]) > 0){  
            largest = i;  
        }  
    } //end for  
    return largest;  
} // end indexOfLargest
```

Tiempo de Ejecución

Veamos el tiempo de ejecución del selection:



The diagram illustrates the time complexity of selection sort. It features three callout boxes: 'Inicialización' pointing to the constant c , 'For' pointing to the summation symbol, and 'indexOf Largest es $O(i)$ ' pointing to the term i in the summation.

$$T_{sel}(n) = c + \sum_{i=1}^{n-1} (c' + i)$$
$$= c + (n - 1) * c' + \frac{n^2}{2} - \frac{n}{2}$$
$$\in \Theta(n^2)$$

Intercambios y Comparaciones

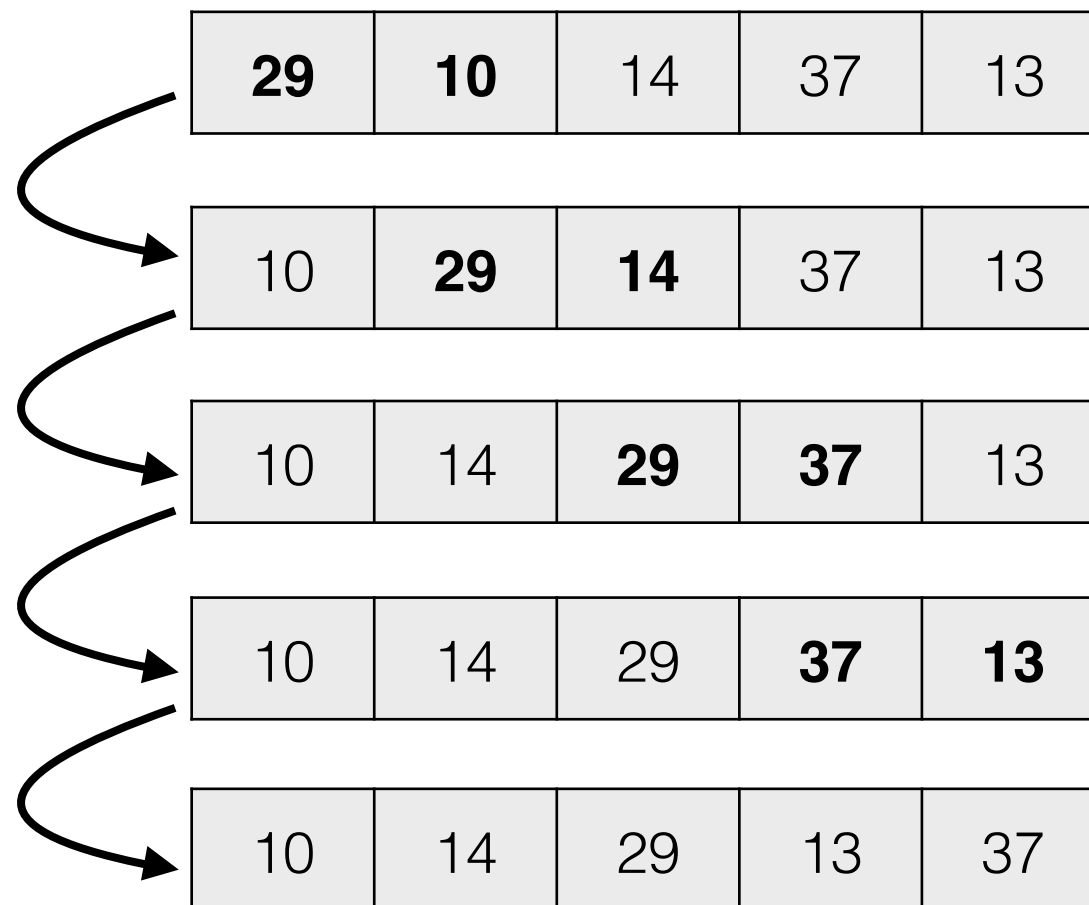
Selection Sort efectua:

- $3 * (n - 1) \in \Theta(n)$ intercambios,
- $\sum_{i=1}^{n-1} i = \frac{n^2}{2} - \frac{n}{2} \in \Theta(n^2)$, comparaciones.

Selection puede ser implementado para que sea estable, la versión de más arriba no lo es.

BubbleSort

Idea: En la primera pasada comparar cada elemento con el siguiente, en caso que no estén ordenados intercambiarlos. Esto pasa el más grande al último. Se hacen N pasadas.



BubbleSort en JAVA

```
public static void bubbleSort(Comparable[] array, int n){
    boolean sorted = false;
    for (int pass = 1; (pass < n)&& !sorted; ++pass){
        // inv: array[n-pass] hasta array[n-1] está ordenado
        sorted = true;
        for (int index = 0; index < n - pass; ++index){
            // inv: para todo 0<=i<index: array[i] <= array[index]
            int nextIndex = index + 1;
            if (array[index].compareTo(array[nextIndex])>0){
                swap(array, index, nextIndex);
                sorted = false;
            } // end if
        } //end for
    } //end for
} // end bubbleSort
```

Tiempo de Ejecución

Veamos su tiempo de ejecución:

$$\begin{aligned}T_{bub}(n) &= \sum_{i=1}^{n-1} \left(c + \sum_{j=0}^{n-i-1} c' \right) \\&= \sum_{i=1}^{n-1} c + \sum_{i=1}^{n-1} \sum_{j=0}^{n-i-1} c' \\&= (n-1) * c + \sum_{i=1}^{n-1} (n-i) * c' \\&= (n-1) * c + c' * n^2 - c' n - c' * \frac{n^2}{2} + c' * \frac{n}{2} \in \Theta(n^2)\end{aligned}$$

Intercambios y Comp.

BubbleSort efectua:

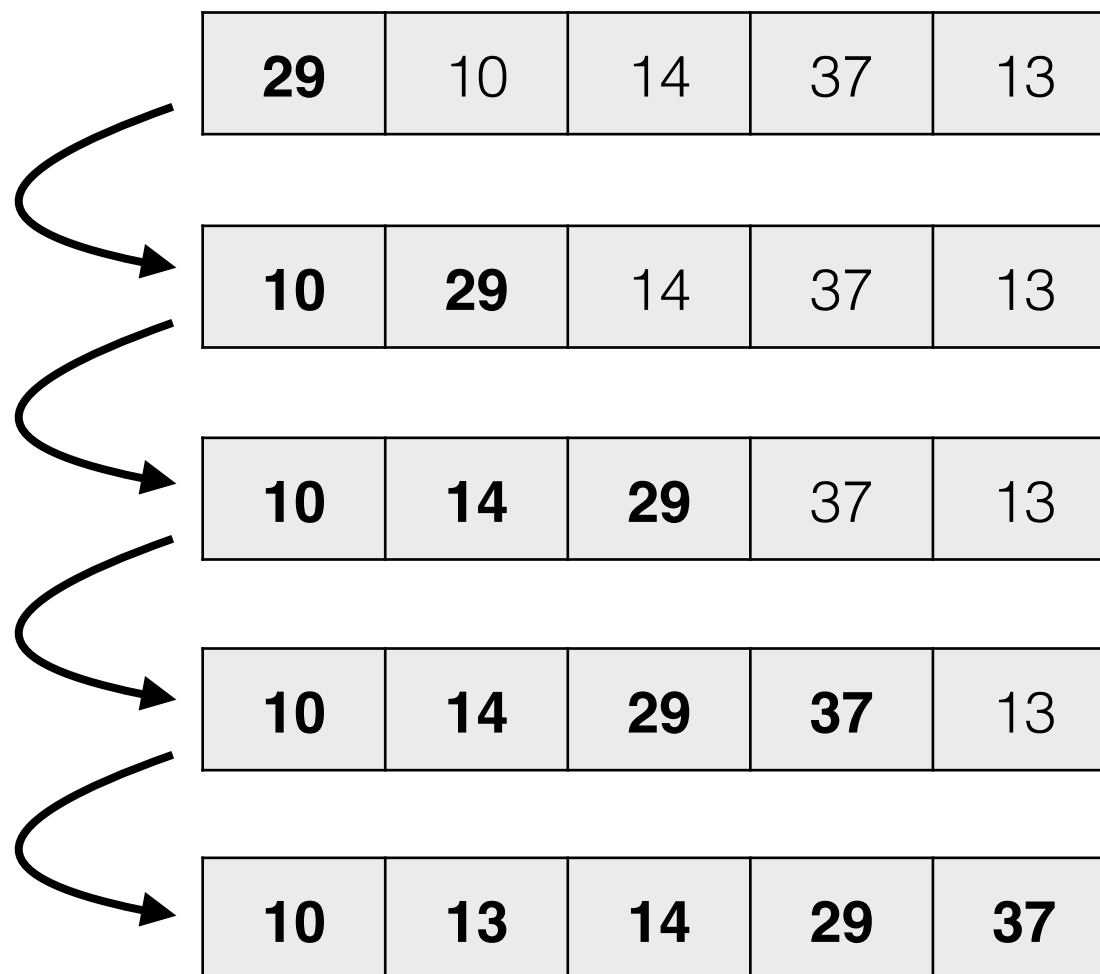
- $(n - 1) + (n - 2) + (n - 3) + \dots \in \Theta(n^2)$ intercambios,
- $(n - 1) + (n - 2) + (n - 3) + \dots \in \Theta(n^2)$ comparaciones.

Además:

- Produce más intercambios que el selection sort.
- Es un algoritmo estable.
- En la práctica no se usa, es uno de los algoritmos más ineficientes

Insertion Sort

Idea: Es el método que usamos cuando jugamos a las cartas. Agarramos un número lo ponemos en su posición, y repetimos.



Insertion en JAVA

```
public static void insertionSort (Comparable [] array , int n){  
    for ( int unsorted = 1; unsorted < n; unsorted++){  
        // array [0.. unsorted -1] esta ordenado  
        Comparable nextItem = array [ unsorted ];  
        int loc = unsorted ;  
        while ((loc > 0) && (array[loc-1].compareTo(nextItem) > 0)){  
            array[loc] = array[loc-1];  
            loc--;  
        }//end while  
        array [ loc ] = nextItem ;  
    }//end for  
}//end insertionSort
```

Tiempo de Ejecución

Analicemos el tiempo de ejecución:

$$\begin{aligned}T_{ins}(n) &= \sum_{i=1}^{n-1} \sum_{j=1}^i c \\&= c * \sum_{i=1}^{n-1} i \\&= c * \frac{n^2}{2} - c * \frac{n}{2} \in \Theta(n^2)\end{aligned}$$

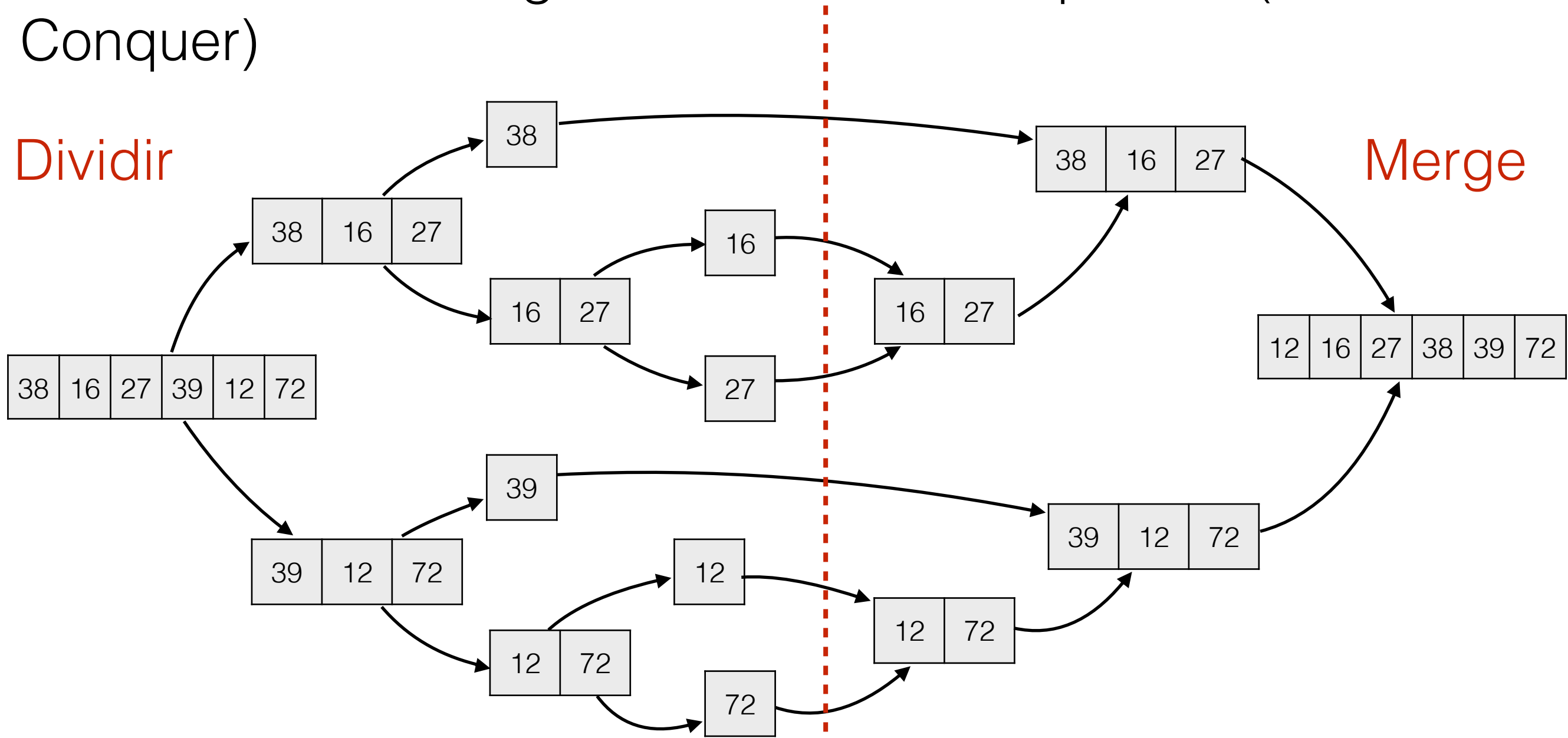
Otras consideraciones

- En el mejor caso es $\Theta(n)$
- La cantidad de comparaciones es: $\Theta(n^2)$
- La cantidad de intercambios: $\Theta(n^2)$
- Insertion Sort es **estable**

Es muy ineficiente para ser usado en la practica.

Mergesort

Idea: Dividimos el arreglo a la mitad y ordenamos recursivamente luego mezclamos las partes (Divide and Conquer)



Tiempo de Ejecución

Analicemos su tiempo de ejecución en el peor caso:

- Merge se puede implementar en $\Theta(n)$
- Su ecuación de recurrencia viene dada por:

$$T(0) = 1$$

$$T(1) = 1$$

$$T(n) = 2 * T\left(\frac{n}{2}\right) + n$$

llamadas recursivas

Merge

Tiempo del MergeSort

Hagamos sustituciones:

$$\begin{aligned} & 2 * T\left(\frac{n}{2}\right) + n \\ &= 2 * \left[2 * T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n \\ &= 2 * \left[2 * \left[2 * T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + \frac{n}{2}\right] + n \\ &= \dots \end{aligned}$$

En i sustituciones nos da: $2^i T\left(\frac{n}{2^i}\right) + i * n$

Obtenemos que: $T\left(\frac{n}{2^i}\right) = 1$ cuando: $\frac{n}{2^i} = 1$ Es decir: $i = \log_2 n$

Reemplazando: $2^{\log_2 n} + 1 + n * \log_2 n = n + n * \log_2 n \in \Theta(n * \log_2 n)$

Mergesort

- Es un algoritmo estable (bien implementado).
- La cantidad de comparaciones es: $O(n * \log n)$
- La cantidad de intercambios es: $O(n * \log n)$

Para entradas pequeñas la recursión hace que no se comporte tan bien.

QuickSort

La idea del quicksort es la siguiente:

- Elegir un pivot (elemento del arreglo).
- Ordenar todos los menores o iguales al pivot antes que el,
- Ordenar todos los mayores después de el
- En ese momento el pivot queda en el lugar que va, se llama recursivamente con la parte a la izq. del pivot y la parte a la derecha.

QuickSort en JAVA

```
public static void quickSort(Comparable[] array, int begin, int end){
    if (begin < end){
        // Calculo la particion
        int p = partition(array, begin, end);
        // ordeno la parte izq
        quickSort(array, begin, p);
        // ordeno la parte derecha
        quickSort(array, p+1, end);
    }
}
```

```
private static int partition(Comparable[] array, int begin, int end){
    Comparable pivot = array[begin];
    int i = begin - 1;
    int j = end + 1;
    while (i < j) {
        //invariante:
        //para k <= i : a[k] <= pivot y para k >= j : pivot <= a[k]
        do j--; while (array[j].compareTo(pivot) > 0);
        do i++; while (array[i].compareTo(pivot) < 0);
        if (i < j) {swap(array, i, j);}
    }
    return j;
}
```

Tiempo de Ejecución

En el peor caso (arreglo ordenado al revés) de elección del pivot, la cantidad de elementos se decrementa por uno, es decir:

$$T(0) = 1$$

$$T(1) = 1$$

$$T(n) = c + T(n - 1) + n$$

tiempo del
partition

Llamada recursiva

Es decir, tenemos: $T(n) \in O(n^2)$

Tiempo en Caso Promedio

En caso promedio tenemos que tomar todas las posibles elecciones del pivot, dividido la longitud de la lista:

$$\begin{aligned}T(0) &= 1 \\T(1) &= 1 \\T(n) &= c + \frac{1}{n} * \sum_{i=0}^{n-1} T(i) + T(n-i)\end{aligned}$$

Si resolvemos esta ecuación nos da: $T(n) \in O(n * \log n)$

Observaciones

- En la practica el QuickSort se comporta mejor que otros algoritmos de sorting.
- El partition no necesita espacio extra.
- No es un algoritmo estable.
- El peor caso tiene pocas probabilidades de suceder!

Cota inferior para Algoritmos de Sorting

Tenemos el siguiente resultado para algoritmo de sorting:

Teorema: Cualquier algoritmo de sorting que utilice comparaciones para ordenar es $\Omega(n * \log n)$

Sin embargo, existen algoritmos que utilizan información extra para ordenar, por ejemplo:

- El mayor número que puede aparecer,
- La cantidad de dígitos que pueden tener los números a ordenar.

Counting Sort

Idea: Para cada i determinamos el número de j 's menores a él en el arreglo, y acomodamos a i en el lugar que va. Necesitamos usar arreglos adicionales para esto.

```
public static void countingSort(int [] array , int n, int k){
    int[] b=new Array[n];
    int[] c = new Array[k];
    for (int i = 0; i < n; i++){
        //contamos la cantidad de elementos iguales a array [ i ]
        c[array[i]] = c[array[i]] + 1;
    }
    for (int i = 1; i < k; i++){
        //contamos la cantidad de elementos iguales o menores a array [ i ]
        c[i] = c[i] + c[i-1];
    }
    // ponemos cada elemento array [ i ] en su lugar
    for (int j = n-1; j >= 0; j--){
        b[c[array[j]] - 1] = array[j]; c[array[j]] = c[array[j]] - 1;
    }
}
```

Ejemplo:

Veamos un ejemplo:

Arreglo Inicial:

3	6	4	1	3	4	1	4
---	---	---	---	---	---	---	---

Contamos:

0	2	0	2	3	0	1
---	---	---	---	---	---	---

Arreglo auxiliar c

Acomodamos:

0	2	2	4	7	7	8
---	---	---	---	---	---	---

Ponemos cada
elemento en su lugar:

1	1	3	3	4	4	4	6
---	---	---	---	---	---	---	---

Arreglo b

Tiempo de Ejecución

Veamos el tiempo de ejecución:

Contar
menores o
iguales

$$T(n) = n + n + k \in \Theta(n)$$

Poner los
elementos en su
lugar

Contar
repetidos

Si el k es muy grande, el algoritmo es ineficiente!

Radix Sort

Idea: También se utiliza para las cartas, primero se ordenan por número y después por palo. Hacemos lo mismo pero por dígito.

Arreglo Inicial:

25	57	48	37	12	92	86	33
----	----	----	----	----	----	----	----

Ordeno por último dígito:

	12,92	33		25	86	57,37	48		
--	-------	----	--	----	----	-------	----	--	--

Acomodo:

12	92	33	25	86	57	37	28
----	----	----	----	----	----	----	----

Ordeno por primer dígito:

12	25,28	33	37	57			86	92	
----	-------	----	----	----	--	--	----	----	--

Acomodo:

12	25	28	33	37	57	86	92
----	----	----	----	----	----	----	----